Operational Semantics
Semantics and applications to verification

Xavier Rival

École Normale Supérieure
Program of this first lecture

Operational semantics

Mathematical description of the execution of programs

1. a model of programs: transition systems
   - definition, a small step semantics
   - a few common examples

2. trace semantics: a families of big step semantics
   - finite and infinite executions
   - fixpoint-based definitions
   - notion of compositional semantics
Outline

1 Transition systems and small step semantics
   - Definition and properties
   - Examples

2 Traces semantics

3 Summary
Definition

We will characterize a program by:

- **states**: photography of the program status at an instant of the execution
- **execution steps**: how do we move from one state to the next one

**Definition: transition systems (TS)**

A **transition system** is a tuple \((S, \rightarrow)\) where:

- \(S\) is the **set of states** of the system
- \(\rightarrow \subseteq \mathcal{P}(S \times S)\) is the **transition relation** of the system

**Note:**

- the set of states **may be infinite**
Transition systems: properties of the transition relation

A **deterministic** system is such that a state fully determines the next state

\[ \forall s_0, s_1, s'_1 \in S, (s_0 \rightarrow s_1 \land s_0 \rightarrow s'_1) \implies s_1 = s'_1 \]

Otherwise, a transition system is **non deterministic**, i.e.:

\[ \exists s_0, s_1, s'_1 \in S, s_0 \rightarrow s_1 \land s_0 \rightarrow s'_1 \land s_1 \neq s'_1 \]

**Notes:**

- transition relation \( \rightarrow \) defines atomic execution steps; it is often called **small-step semantics** or **structured operational semantics**
- steps are **discrete** (not continuous) to describe both discrete and continuous behaviors, we would need to look at **hybrid systems** (beyond the scope of this lecture)
Transition systems: special states

**Initial / final states:**
we often consider transition systems with a set of initial and final states:

- a set of **initial states** $S_I \subseteq S$ denotes states where the execution should start
- a set of **final states** $S_F \subseteq S$ denotes states where the execution should reach the end of the program

When needed, we add these to the definition of the transition systems $((S, \rightarrow, S_I, S_F))$.

**Blocking state** (not the same as final state):

- a state $s_0 \in S$ is **blocking** when it is the origin of no transition: $\forall s_1 \in S, \neg(s_0 \rightarrow s_1)$
- example: we often introduce an **error state** (usually noted $\Omega$ to denote the erroneous, blocking configuration)
Outline

1 Transition systems and small step semantics
   - Definition and properties
   - Examples

2 Traces semantics

3 Summary
Finite automata as transition systems

We can clearly formalize the **word recognition** by a finite automaton using a transition system:

- we consider automaton $\mathcal{A} = (Q, q_i, q_f, \rightarrow)$
- a “state” is defined by:
  - the remaining of the word to recognize
  - the automaton state that has been reached so far

  thus, $S = Q \times L^*$

- the **transition relation** $\rightarrow$ of the transition system is defined by:

  $$(q_0, aw) \rightarrow (q_1, w) \iff q_0 \xrightarrow{a} q_1$$

- the **initial** and **final states** are defined by:

  $$S_I = \{(q_i, w) | w \in L^*\} \quad S_F = \{(q_f, \epsilon)\}$$
Pure $\lambda$-calculus

A **bare bones model of functional programming:**

### $\lambda$-terms

The set of $\lambda$-terms is defined by:

\[
\begin{align*}
t, u, \ldots & ::= x \quad \text{variable} \\
             & | \lambda x \cdot t \quad \text{abstraction} \\
             & | t \ u \quad \text{application}
\end{align*}
\]

### $\beta$-reduction

- $(\lambda x \cdot t) u \rightarrow_{\beta} t[x \leftarrow u]$
- if $u \rightarrow_{\beta} v$ then $\lambda x \cdot u \rightarrow_{\beta} \lambda x \cdot v$
- if $u \rightarrow_{\beta} v$ then $u \ t \rightarrow_{\beta} v \ t$
- if $u \rightarrow_{\beta} v$ then $t \ u \rightarrow_{\beta} t \ v$

The $\lambda$-calculus defines a transition system:

- $\mathcal{S}$ is the set of $\lambda$-terms and $\rightarrow_{\beta}$ the transition relation
- $\rightarrow_{\beta}$ is **non-deterministic**; example ?
  though, ML fixes an execution order
- given a lambda term $t_0$, we may consider $(\mathcal{S}, \rightarrow_{\beta}, \mathcal{S}_I)$ where $\mathcal{S}_I = \{t_0\}$
- **blocking states** are terms with no redex $(\lambda x \cdot u) v$
A MIPS like assembly language: syntax

We now consider a (very simplified) **assembly language**

- machine integers: sequences of 32-bits (set: \( \mathbb{B}^{32} \))
- instructions are encoded over 32-bits (set: \( \mathbb{I}_{\text{MIPS}} \))
  and stored into the same space as data (i.e., \( \mathbb{I}_{\text{MIPS}} \subseteq \mathbb{B}^{32} \))

### Memory configurations

- **program counter** \( \text{pc} \)
  - current instruction
- **general purpose registers** \( r_0 \ldots r_{31} \)
- **main memory** (RAM)
  - \( \text{mem} : \text{Addrs} \rightarrow \mathbb{B}^{32} \)
    where \( \text{Addrs} \subseteq \mathbb{B}^{32} \)

### Instructions

\[
i ::= (\in \mathbb{I}_{\text{MIPS}}) \\
| \text{add } r_d, r_{s0}, r_{s1} & \text{addition} \\
| \text{addi } r_d, r_{s0}, v & \text{add. } v \in \mathbb{B}^{32} \\
| \text{sub } r_d, r_{s0}, r_{s1} & \text{subtraction} \\
| b \text{ dst} & \text{branch} \\
| \text{blt } r_{s0}, r_{s1}, dst & \text{cond. branch} \\
| \text{ld } r_d, o, r_x & \text{relative load} \\
| \text{st } r_d, o, r_x & \text{relative store} \\
\]

\( v, dst, o \in \mathbb{B}^{32} \)
Definition: state

A state is a tuple \((pc, \rho, \mu)\) which comprises:

- a **program counter** value \(pc \in \mathbb{B}^{32}\)
- a function mapping each **general purpose register** to its value \(\rho : \{0, \ldots, 31\} \rightarrow \mathbb{B}^{32}\)
- a function mapping each **memory cell** to its value \(\mu : \text{Addrs} \rightarrow \mathbb{B}^{32}\)

What would a **dangerous state** be?

- writing **over an instruction**
- reading or writing **outside the program’s memory**

⇒ we cannot fully formalize these yet...
   as we need to formalize the behavior of each instruction first
We assume a state \( s = (pc, \rho, \mu) \) and that \( \mu(pc) = i \); then:

- if \( i = \text{add} \ r_d, r_{s0}, r_{s1} \), then:
  \[
  s \rightarrow (pc + 4, \rho[d \mapsto (\rho(s0) + \rho(s1))], \mu)
  \]

- if \( i = \text{addi} \ r_d, r_{s0}, v \), then:
  \[
  s \rightarrow (pc + 4, \rho[d \mapsto (\rho(s0) + v)], \mu)
  \]

- if \( i = \text{sub} \ r_d, r_{s0}, r_{s1} \), then:
  \[
  s \rightarrow (pc + 4, \rho[d \mapsto (\rho(s0) - \rho(s1))], \mu)
  \]

- if \( i = b \ dst \), then:
  \[
  s \rightarrow (dst, \rho, \mu)
  \]
A MIPS like assembly language: transition relation

We assume a state \( s = (pc, \rho, \mu) \) and that \( \mu(pc) = i \); then:

- if \( i = \text{blt} \ r_{s0}, r_{s1}, dst \), then:
  \[
  s \rightarrow \begin{cases} 
  (dst, \rho, \mu) & \text{if } \rho(s0) < \rho(s1) \\
  (pc + 4, \rho, \mu) & \text{otherwise}
  \end{cases}
  \]

- if \( i = \text{id} \ r_d, o, r_x \), then:
  \[
  s \rightarrow \begin{cases} 
  (pc + 4, \rho[d \mapsto \mu(\rho(x) + o)], \mu) & \text{if } \mu(\rho(x) + o) \text{ is defined} \\
  \Omega & \text{otherwise}
  \end{cases}
  \]

- if \( i = \text{st} \ r_d, o, r_x \), then:
  \[
  s \rightarrow \begin{cases} 
  (pc + 4, \rho, \mu[\rho(x) + o] \mapsto \rho(d)]) & \text{if } \mu(\rho(x) + o) \text{ is defined} \\
  \Omega & \text{otherwise}
  \end{cases}
  \]
A simple imperative language: syntax

We now look at a more classical imperative language (intuitively, a bare-bone subset of C):

- **variables** $X$: finite, predefined set of variables
- **labels** $L$: before and after each statement
- **values** $V$: $V_{\text{int}} \cup V_{\text{float}} \cup \ldots$

**Syntax**

```plaintext
e ::= v ∈ $V_{\text{int}} \cup V_{\text{float}} \cup \ldots$ | e + e | e * e | ... expressions
c ::= TRUE | FALSE | e < e | e = e conditions
i ::= x := e;
   | if(c) b else b assignment
   | while(c) b condition
b ::= {i;...;i;} loop
```

block, program($\mathbb{P}$)
A non-error state should fully describe the configuration at one instant of the program execution:

- the **memory state** defines the current contents of the memory
  \[ m \in M = X \rightarrow \mathbb{V} \]
- the **control state** defines *where* the program currently is
  - analogous to the program counter
  - can be defined by adding **labels** \( \mathbb{L} = \{ \ell_0, \ell_1, \ldots \} \) between each pair of consecutive statements; then:
    \[ S = L \times M \cup \{ \Omega \} \]
  - or by the program remaining to be executed; then:
    \[ S = P \times M \cup \{ \Omega \} \]
A simple imperative language: semantics of expressions

- The **semantics** $\llbracket e \rrbracket$ of expression $e$ should evaluate each expression into a value, given a memory state.
- **Evaluation errors** may occur: division by zero... error value is also noted $\Omega$

Thus: $\llbracket e \rrbracket : M \rightarrow \mathbb{V} \cup \{\Omega\}$

**Definition**, by **induction over the syntax**:

$$
\begin{align*}
\llbracket v \rrbracket (m) &= v \\
\llbracket x \rrbracket (m) &= m(x) \\
\llbracket e_0 + e_1 \rrbracket (m) &= \llbracket e_0 \rrbracket (m) \oplus \llbracket e_1 \rrbracket (m) \\
\llbracket e_0 / e_1 \rrbracket (m) &= \begin{cases} 
\Omega & \text{if } \llbracket e_1 \rrbracket (m) = 0 \\
\llbracket e_0 \rrbracket (m) / \llbracket e_1 \rrbracket (m) & \text{otherwise}
\end{cases}
\end{align*}
$$

where $\oplus$ is the machine implementation of operator $\oplus$, and is $\Omega$-strict, i.e., $\forall v \in \mathbb{V}, \; v \oplus \Omega = \Omega \oplus v = \Omega$. 

A simple imperative language: semantics of conditions

- The semantics $\llbracket c \rrbracket$ of condition $c$ should return a boolean value.
- It follows a similar definition to that of the semantics of expressions:
  $\llbracket c \rrbracket : M \rightarrow \mathbb{V}_{\text{bool}} \cup \{\Omega\}$

**Definition**, by induction over the syntax:

- $\llbracket \text{TRUE} \rrbracket (m) = \text{TRUE}$
- $\llbracket \text{FALSE} \rrbracket (m) = \text{FALSE}$
- $\llbracket e_0 < e_1 \rrbracket (m) = \begin{cases} \text{TRUE} & \text{if } \llbracket e_0 \rrbracket (m) < \llbracket e_1 \rrbracket (m) \\ \text{FALSE} & \text{if } \llbracket e_0 \rrbracket (m) \geq \llbracket e_1 \rrbracket (m) \\ \Omega & \text{if } \llbracket e_0 \rrbracket (m) = \Omega \text{ or } \llbracket e_1 \rrbracket (m) = \Omega \end{cases}$
- $\llbracket e_0 = e_1 \rrbracket (m) = \begin{cases} \text{TRUE} & \text{if } \llbracket e_0 \rrbracket (m) = \llbracket e_1 \rrbracket (m) \\ \text{FALSE} & \text{if } \llbracket e_0 \rrbracket (m) \neq \llbracket e_1 \rrbracket (m) \\ \Omega & \text{if } \llbracket e_0 \rrbracket (m) = \Omega \text{ or } \llbracket e_1 \rrbracket (m) = \Omega \end{cases}$
A simple imperative language: transitions

We now consider the transition induced by each statement.

Case of **assignment** $l_0 : x = e; l_1$
- If $\llbracket e \rrbracket(m) \neq \Omega$, then $(l_0, m) \rightarrow (l_1, m[x \leftarrow \llbracket e \rrbracket(m)])$
- If $\llbracket e \rrbracket(m) = \Omega$, then $(l_0, m) \rightarrow \Omega$

Case of **condition** $l_0 : \text{if}(c)\{l_1 : b_t \ l_2\} \text{else}\{l_3 : b_f \ l_4\} \ l_5$
- If $\llbracket c \rrbracket(m) = \text{TRUE}$, then $(l_0, m) \rightarrow (l_1, m)$
- If $\llbracket c \rrbracket(m) = \text{FALSE}$, then $(l_0, m) \rightarrow (l_3, m)$
- If $\llbracket c \rrbracket(m) = \Omega$, then $(l_0, m) \rightarrow \Omega$
- $(l_2, m) \rightarrow (l_5, m)$
- $(l_4, m) \rightarrow (l_5, m)$
A simple imperative language: transitions

Case of **loop** $l_0 : \textbf{while}(c)\{l_1 : b_t l_2\} l_3$

- if $\llbracket c \rrbracket(m) = \text{TRUE}$, then $\begin{cases} (l_0, m) \rightarrow (l_1, m) \\ (l_2, m) \rightarrow (l_1, m) \end{cases}$
- if $\llbracket c \rrbracket(m) = \text{FALSE}$, then $\begin{cases} (l_0, m) \rightarrow (l_3, m) \\ (l_2, m) \rightarrow (l_3, m) \end{cases}$
- if $\llbracket c \rrbracket(m) = \Omega$, then $\begin{cases} (l_0, m) \rightarrow \Omega \\ (l_2, m) \rightarrow \Omega \end{cases}$

Case of $\{l_0 : i_0; l_1 : \ldots; l_{n-1}i_{n-1}; l_n\}$

- the transition relation is defined by the individual instructions
Extending the language with non-determinism

The language we have considered so far is a bit **limited**:

- it is **deterministic**: at most one transition possible from any state
- it does not support the **input of values**

**Changes if we model non-deterministic inputs...**

**... with an input instruction:**

- \( i ::= \ldots | x ::= \text{input()} \)
- \( l_0 : x ::= \text{input()} ; l_1 \) generates transitions
  \[
  \forall v \in \mathbb{V}, (l_0, m) \rightarrow (l_1, m[x \leftarrow v])
  \]
- one instruction induces non-determinism

**... with a random function:**

- \( e ::= \ldots | x ::= \text{rand()} \)
- **expressions** have a **non-deterministic** semantics:
  \[
  \llbracket e \rrbracket : M \rightarrow \mathcal{P}(\mathbb{V} \cup \{\Omega\})
  \]
  \[
  \llbracket \text{rand()} \rrbracket (m) = \mathbb{V}
  \]
  \[
  \llbracket v \rrbracket (m) = \{v\}
  \]
  \[
  \llbracket c \rrbracket : M \rightarrow \mathcal{P}(\mathbb{V}_{\text{bool}} \cup \{\Omega\})
  \]
- all instructions induce non-determinism
Semantics of real world programming languages

C language:
- several norms: ANSI C’99, ANSI C’11, K&R...
- not fully specified:
  - undefined behavior
  - implementation dependent behavior: architecture (ABI) or implementation (compiler...)
  - unspecified parts: leave room for implementation of compilers and optimizations
- formalizations in HOL (C’99), in Coq (CompCert C compiler)

OCaml language:
- more formal...
- ... but still with some unspecified parts, e.g., execution order
Outline

1 Transition systems and small step semantics

2 Traces semantics
   - Definitions
   - Finite traces semantics
   - Fixpoint definition
   - Compositionality
   - Infinite traces semantics

3 Summary
Execution traces

- So far, we considered only states and atomic transitions
- We now consider program executions as a whole

**Definition: traces**

- A **finite trace** is a finite sequence of states \(s_0, \ldots, s_n\), noted \(\langle s_0, \ldots, s_n \rangle\)
- An **infinite trace** is an infinite sequence of states \(\langle s_0, \ldots \rangle\)

Besides, we write:

- \(S^*\) for the set of finite traces
- \(S^\omega\) for the set of infinite traces
- \(S^\infty = S^* \cup S^\omega\) for the set of finite or infinite traces
Operations on traces: concatenation

**Definition: concatenation**

The **concatenation operator** \( \cdot \) is defined by:

\[
\langle s_0, \ldots, s_n \rangle \cdot \langle s'_0, \ldots, s'_n \rangle = \langle s_0, \ldots, s_n, s'_0, \ldots, s'_n \rangle \\
\langle s_0, \ldots, s_n \rangle \cdot \langle s'_0, \ldots \rangle = \langle s_0, \ldots, s_n, s'_0, \ldots \rangle \\
\langle s_0, \ldots, s_n, \ldots \rangle \cdot \sigma' = \langle s_0, \ldots, s_n, \ldots \rangle
\]

We also define:

- the **empty trace** \( \varepsilon \), neutral element for \( \cdot \).
- the **length** operator \( |.| \):

\[
\begin{cases}
|\varepsilon| = 0 \\
|\langle s_0, \ldots, s_n \rangle| = n + 1 \\
|\langle s_0, \ldots \rangle| = \omega
\end{cases}
\]
Comparing traces: the prefix order relation

**Definition: prefix order relation**

Relation $\prec$ is defined by:

\[
\langle s_0, \ldots, s_n \rangle \prec \langle s'_0, \ldots, s'_n \rangle \iff \begin{cases} 
    n \leq n' \\
    \forall i \in \llbracket 0, n \rrbracket, \ s_i = s'_i
\end{cases}
\]

\[
\langle s_0, \ldots \rangle \prec \langle s'_0, \ldots \rangle \iff \forall i \in \mathbb{N}, \ s_i = s'_i
\]

\[
\langle s_0, \ldots, s_n \rangle \prec \langle s'_0, \ldots \rangle \iff \forall i \in \llbracket 0, n \rrbracket, \ s_i = s'_i
\]

Proof: straightforward application of the definition of order relations
Outline

1 Transition systems and small step semantics

2 Traces semantics
   - Definitions
   - Finite traces semantics
   - Fixpoint definition
   - Compositionality
   - Infinite traces semantics

3 Summary
Semantics of finite traces

We consider a transition system $S = (S, \rightarrow)$

**Definition**

The *finite traces semantics* $[S]^*$ is defined by:

$$[S]^* = \{ \langle s_0, \ldots, s_n \rangle \in S^* \mid \forall i, s_i \rightarrow s_{i+1} \}$$

**Example:**

- contrived transition system $S = (\{a, b, c, d\}, \{(a, b), (b, a), (b, c)\})$
- finite traces semantics:

$$[S]^* = \{ \langle a, b, \ldots, a, b, a \rangle, \langle b, a, \ldots, a, b, a \rangle, \langle a, b, \ldots, a, b, a, b \rangle, \langle b, a, \ldots, a, b, a, b \rangle, \langle a, b, \ldots, a, b, a, b, c \rangle, \langle b, a, \ldots, a, b, a, b, c \rangle, \langle c \rangle, \langle d \rangle \}$$
Interesting subsets of the finite trace semantics

We consider a transition system $S = (\mathcal{S}, \rightarrow, \mathcal{S}_I, \mathcal{S}_F)$

- **the initial traces**, i.e., starting from an initial state:
  \[
  \{\langle s_0, \ldots, s_n \rangle \in [\mathcal{S}]^* \mid s_0 \in \mathcal{S}_I\}
  \]

- **the traces reaching a blocking state**:
  \[
  \{\sigma \in [\mathcal{S}]^* \mid \forall \sigma' \in [\mathcal{S}]^*, \sigma \prec \sigma' \implies \sigma = \sigma'\}
  \]

- **the traces ending in a final state**:
  \[
  \{\langle s_0, \ldots, s_n \rangle \in [\mathcal{S}]^* \mid s_n \in \mathcal{S}_F\}
  \]

**Example** (same transition system, with $\mathcal{S}_I = \{a\}$ and $\mathcal{S}_F = \{c\}$):
- traces from an initial state ending in a final state:
  \[
  \{\langle a, b, \ldots, a, b, a, b, c \rangle\}
  \]
Example: finite automaton

We consider the example of the previous course:

\[ L = \{a, b\} \quad Q = \{q_0, q_1, q_2\} \]
\[ q_i = q_0 \quad q_f = q_2 \]
\[ q_0 \xrightarrow{a} q_1 \quad q_1 \xrightarrow{b} q_2 \quad q_2 \xrightarrow{a} q_1 \]

Then, we have the following traces:

\[ \tau_0 = \langle (q_0, ab), (q_1, b), (q_2, \epsilon) \rangle \]
\[ \tau_1 = \langle (q_0, abab), (q_1, bab), (q_2, ab), (q_1, b), (q_2, \epsilon) \rangle \]
\[ \tau_2 = \langle (q_0, ababab), (q_1, babab), (q_2, abab), (q_1, bab) \rangle \]
\[ \tau_3 = \langle (q_0, abaaa), (q_1, baaa), (q_2, aaa), (q_1, aa) \rangle \]

Then:
- \( \tau_0, \tau_1 \) are initial traces, reaching a final state
- \( \tau_2 \) is an initial trace, and is not maximal
- \( \tau_3 \) reaches a blocking state, but not a final state
Example: $\lambda$-term

We consider $\lambda$-term $\lambda y \cdot (((\lambda x \cdot y)((\lambda x \cdot x x)(\lambda x \cdot x x))))$, and show two traces generated from it (at each step the reduced lambda is shown in red):

$$\tau_0 = \langle \lambda y \cdot (((\lambda x \cdot y)((\lambda x \cdot x x)(\lambda x \cdot x x))))\lambda y \cdot y \rangle$$

$$\tau_1 = \langle \lambda y \cdot (((\lambda x \cdot y)((\lambda x \cdot x x)(\lambda x \cdot x x)))) \lambda y \cdot (((\lambda x \cdot y)((\lambda x \cdot x x)(\lambda x \cdot x x)))) \lambda y \cdot (((\lambda x \cdot y)((\lambda x \cdot x x)(\lambda x \cdot x x)))) \lambda y \cdot (((\lambda x \cdot y)((\lambda x \cdot x x)(\lambda x \cdot x x)))) \rangle$$

Then:

- $\tau_0$ is a maximal traces; it reaches a final state (no more reduction can be done)
- $\tau_1$ can be extended for arbitrarily many steps; the second part of the course will study infinite traces
Example: imperative program

Similarly, we can write the traces of a simple imperative program:

$l_0$: \( x := 1; \)
$l_1$: \( y := 0; \)
$l_2$: \( \textbf{while}(x < 4)\{
\)
\hspace{1em} \( y := y + x; \)
\hspace{1em} \( x := x + 1; \)
\hspace{1em} \} \)
$l_5$: \( \) \( \) \( \) \( \) \( \) \( \) \( \)
$l_6$: \( \) \( \text{(final program point)} \)

\[ \tau = \langle (l_0, (x = 6, y = 8)), (l_1, (x = 1, y = 8)), (l_2, (x = 1, y = 0)), (l_3, (x = 1, y = 0)), (l_4, (x = 1, y = 1)), (l_5, (x = 2, y = 1)), (l_3, (x = 2, y = 1)), (l_4, (x = 2, y = 3)), (l_5, (x = 3, y = 3)), (l_3, (x = 3, y = 3)), (l_4, (x = 3, y = 6)), (l_5, (x = 4, y = 6)), (l_6, (x = 4, y = 6)) \rangle \]

- very \textit{precise} description of what the program does...
- ... but \textit{quite cumbersome}
Outline

1 Transition systems and small step semantics

2 Traces semantics
   - Definitions
   - Finite traces semantics
   - Fixpoint definition
   - Compositionality
   - Infinite traces semantics

3 Summary
Towards a fixpoint definition

We consider again our contrived transition system

\[ S = (\{a, b, c, d\}, \{(a, b), (b, a), (b, c)\}) \]

Traces by length:

<table>
<thead>
<tr>
<th>(i)</th>
<th>traces of length (i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\epsilon)</td>
</tr>
<tr>
<td>1</td>
<td>(\langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle)</td>
</tr>
<tr>
<td>2</td>
<td>(\langle a, b \rangle, \langle b, a \rangle, \langle b, c \rangle)</td>
</tr>
<tr>
<td>3</td>
<td>(\langle a, b, a \rangle, \langle b, a, b \rangle, \langle a, b, c \rangle)</td>
</tr>
<tr>
<td>4</td>
<td>(\langle a, b, a, b \rangle, \langle b, a, b, a \rangle, \langle b, a, b, c \rangle)</td>
</tr>
</tbody>
</table>

Like the automaton in lecture 1, this suggests a least fixpoint definition: traces of length \(i + 1\) can be derived from the traces of length \(i\), by adding a transition
Trace semantics fixpoint form

We define a **semantic function**, that computes the traces of length $i + 1$ from the traces of length $i$ (where $i \geq 1$):

**Finite traces semantics as a fixpoint**

Let $\mathcal{I} = \{\epsilon\} \cup \{\langle s \rangle \mid s \in S\}$.
Let $F_\star$ be the function defined by:

$$F_\star : \mathcal{P}(S^\star) \longrightarrow \mathcal{P}(S^\star)$$

$$X \longmapsto X \cup \{\langle s_0, \ldots, s_n, s_{n+1} \rangle \mid \langle s_0, \ldots, s_n \rangle \in X \land s_n \rightarrow s_{n+1}\}$$

Then, $F_\star$ is **continuous** and thus has a least-fixpoint greater than $\mathcal{I}$; moreover:

$$\text{lfp}_\mathcal{I} F_\star = [S]^\star = \bigcup_{n \in \mathbb{N}} F_\star^n(\mathcal{I})$$
Fixpoint definition: proof (1), fixpoint existence

First, we prove that $F_\star$ is **continuous**.

Let $\mathcal{X} \subseteq \mathcal{P}(\mathcal{S}^*)$ and $A = \bigcup_{X \in \mathcal{X}} X$. Then:

\[
F_\star(\bigcup_{X \in \mathcal{X}} X) \\
= A \cup \{\langle s_0, \ldots, s_n, s_{n+1}\rangle \mid (\langle s_0, \ldots, s_n\rangle \in \bigcup_{X \in \mathcal{X}} X) \land s_n \to s_{n+1}\} \\
= A \cup \{\langle s_0, \ldots, s_n, s_{n+1}\rangle \mid (\exists X \in \mathcal{X}, \langle s_0, \ldots, s_n\rangle \in X) \land s_n \to s_{n+1}\} \\
= A \cup \{\langle s_0, \ldots, s_n, s_{n+1}\rangle \mid \exists X \in \mathcal{X}, \langle s_0, \ldots, s_n\rangle \in X \land s_n \to s_{n+1}\} \\
= \left(\bigcup_{X \in \mathcal{X}} X\right) \cup \left(\bigcup_{X \in \mathcal{X}} \{\langle s_0, \ldots, s_n, s_{n+1}\rangle \mid \langle s_0, \ldots, s_n\rangle \in X \land s_n \to s_{n+1}\}\right) \\
= \bigcup_{X \in \mathcal{X}} \left(X \cup \{\langle s_0, \ldots, s_n, s_{n+1}\rangle \mid \langle s_0, \ldots, s_n\rangle \in X \land s_n \to s_{n+1}\}\right) \\
= \bigcup_{X \in \mathcal{X}} F_\star(X)
\]

Function $F_\star$ is $\cup$-complete, hence continuous.

As $(\mathcal{P}(\mathcal{S}^*), \subseteq)$ is a CPO, the continuity of $F_\star$ entails the **existence of a least-fixpoint** (Kleene theorem); moreover, it implies that:

\[
\text{Lfp}_{\mathcal{I}} F_\star = \bigcup_{n \in \mathbb{N}} F_\star^n(\mathcal{I})
\]
Fixpoint definition: proof (2), fixpoint equality

We now show that $[S]^*$ is equal to $\text{lfp}_\mathcal{I}F_\star$, by showing the property below, by induction over $n$:

$$\forall k \leq n, \langle s_0, \ldots, s_k \rangle \in F^n_\star(\mathcal{I}) \iff \langle s_0, \ldots, s_k \rangle \in [S]^*$$

- at rank 0, only traces of length 1 need be considered:
  $$\langle s \rangle \in [S]^* \iff s \in S \iff \langle s \rangle \in F^0_\star(\mathcal{I})$$

- at rank $n + 1$, and assuming the property holds at rank $n$ (the equivalence is obvious for traces of length 1):
  $$\langle s_0, \ldots, s_k, s_{k+1} \rangle \in [S]^*$$
  $$\iff \langle s_0, \ldots, s_k \rangle \in [S]^* \land s_k \rightarrow s_{k+1}$$
  $$\iff \langle s_0, \ldots, s_k \rangle \in F^n_\star(\mathcal{I}) \land s_k \rightarrow s_{k+1} \quad (k \leq n \text{ since } k + 1 \leq n + 1)$$
  $$\iff \langle s_0, \ldots, s_k, s_{k+1} \rangle \in F^{n+1}_\star(\mathcal{I})$$
Trace semantics fixpoint form: example

Example, with the same simple transition system $S = (S, \rightarrow)$:

- $S = \{a, b, c, d\}$
- $\rightarrow$ is defined by $a \rightarrow b$, $b \rightarrow a$ and $b \rightarrow c$

Then, the first iterates are:

\[
\begin{align*}
F^0_\star(I) &= \{\epsilon, \langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle\} \\
F^1_\star(I) &= F^0_\star(I) \cup \{\langle b, a \rangle, \langle a, b \rangle, \langle b, c \rangle\} \\
F^2_\star(I) &= F^1_\star(I) \cup \{\langle a, b, a \rangle, \langle b, a, b \rangle, \langle a, b, c \rangle\} \\
F^3_\star(I) &= F^2_\star(I) \cup \{\langle b, a, b, a \rangle, \langle a, b, a, b \rangle, \langle b, a, b, c \rangle\} \\
F^4_\star(I) &= F^3_\star(I) \cup \{\langle a, b, a, b, a \rangle, \langle b, a, b, a, b \rangle, \langle a, b, a, b, c \rangle\} \\
F^5_\star(I) &= \ldots
\end{align*}
\]

- the traces of $\llbracket S \rrbracket^*$ of length $n + 1$ appear in $F^n_\star(I)$
Outline

1 Transition systems and small step semantics

2 Traces semantics
   - Definitions
   - Finite traces semantics
   - Fixpoint definition
   - Compositionality
   - Infinite traces semantics

3 Summary
Notion of compositional semantics

The traces semantics definition we have seen is **global**:
- the **whole system** defines a **transition relation**
- we **iterate** this relation until we get a fixpoint

Though, a **modular** definition would be nicer, to allow reasoning on program fragments, or derive properties of a program from properties of its pieces...

**Can we derive a more modular expression of the semantics?**
Traces semantics  Compositionality

Notion of compositional semantics

Observation: programs often have an inductive structure

- \(\lambda\)-terms are defined by induction over the syntax
- imperative programs are defined by induction over the syntax
- there are exceptions: our MIPS language does not naturally look that way

Definition: compositional semantics

A semantics \(\llbracket . \rrbracket\) is said to be \textbf{compositional} when the semantics of a program can be defined as a function of the semantics of its parts, i.e.,

When program \(\pi\) writes down as \(C[\pi_0, \ldots, \pi_k]\) where \(\pi_0, \ldots, \pi_k\) are its components, there exists a function \(F_C\) such that

\[
\llbracket \pi \rrbracket = F_C(\llbracket \pi_0 \rrbracket, \ldots, \llbracket \pi_k \rrbracket),
\]

where \(F_C\) depends only on syntactic construction \(F_C\).
Case of a simplified imperative language

Case of a sequence of two instructions \( b \equiv l_0 : i_0; l_1 : i_1; l_2 : \)

\[
[b]^* = [i_0]^* \cup [i_1]^*
\]

\[
\cup \{ \langle s_0, \ldots, s_m \rangle \mid \exists n \in [0, m], \\
\langle s_0, \ldots, s_n \rangle \in [i_0]^* \land \langle s_n, \ldots, s_m \rangle \in [i_1]^* \}
\]

This amounts to **concatenating** traces of \([i_0]^*\) and \([i_1]^*\) that share a state in common (necessarily at point \(l_1\)).

Cases of a condition, a loop: similar

- by **concatenation** of traces around junction points
- by doing a **least-fixpoint computation** over loops

We can provide a compositional semantics for our simplified imperative language
Case of \( \lambda \)-calculus

Case of a \( \lambda \)-term \( t = (\lambda x \cdot u) \, v \):

- executions may start with a reduction in \( u \)
- executions may start with a reduction in \( v \)
- executions may start with the reduction of the head redex
- an execution may mix reductions steps in \( u \) and \( v \) in an arbitrary order

No nice compositional trace semantics of \( \lambda \)-calculus...
Outline

1. Transition systems and small step semantics

2. Traces semantics
   - Definitions
   - Finite traces semantics
   - Fixpoint definition
   - Compositionality
   - Infinite traces semantics

3. Summary
Non termination

Can the finite traces semantics express non termination?

Consider the case of our contrived system:

$$S = \{a, b, c, d\} \quad \rightarrow = \{(a, b), (b, a), (b, c)\}$$

- this system clearly has non-terminating behaviors: it can loop from $a$ to $b$ and back forever
- the finite traces semantics does show the existence of this cycle as there exists an infinite chain of finite traces for the prefix order $\preceq$:
  $$\langle a, b \rangle, \langle a, b, a \rangle, \langle a, b, a, b \rangle, \langle a, b, a, b, a \rangle, \ldots \in [S]^*$$
- though, the existence of this chain is not very obvious

Thus, we now define a semantics made of infinite traces
Semantics of infinite traces

We consider a transition system $S = (S, \rightarrow)$

**Definition**

The **infinite traces semantics** $\llbracket S \rrbracket^\omega$ is defined by:

$$\llbracket S \rrbracket^\omega = \{ \langle s_0, \ldots \rangle \in S^\omega \mid \forall i, s_i \rightarrow s_{i+1} \}$$

**Infinite traces starting from an initial state** (considering $S = (S, \rightarrow, S_I, S_F)$):

$$\{ \langle s_0, \ldots \rangle \in \llbracket S \rrbracket^\omega \mid s_0 \in S_I \}$$

**Example:**

- contrived transition system defined by
  $$S = \{ a, b, c, d \} \quad (\rightarrow) = \{(a, b), (b, a), (b, c)\}$$

- the infinite traces semantics contains **exactly two** traces
  $$\llbracket S \rrbracket^\omega = \{ \langle a, b, \ldots, a, b, a, b, \ldots \rangle, \langle b, a, \ldots, b, a, b, a, \ldots \rangle \}$$
Fixpoint form

Can we also provide a fixpoint form for $\llbracket S \rrbracket^\omega$?

Intuitively, $\langle s_0, s_1, \ldots \rangle \in \llbracket S \rrbracket^\omega$ if and only if $\forall n, s_n \rightarrow s_{n+1}$, i.e.,

$$\forall n \in \mathbb{N}, \forall k \leq n, s_k \rightarrow s_{k+1}$$

Let $F_\omega$ be defined by:

$$F_\omega : \mathcal{P}(S^\omega) \rightarrow \mathcal{P}(S^\omega)$$

$$X \mapsto \{ \langle s_0, s_1, \ldots, s_n, \ldots \rangle \mid \langle s_1, \ldots, s_n, \ldots \rangle \in X \land s_0 \rightarrow s_1 \}$$

Then, we can show by induction that:

$$\sigma \in \llbracket S \rrbracket^\omega \iff \forall n \in \mathbb{N}, \sigma \in F^n_\omega(S^\omega)$$

$$\iff \bigcap_{n \in \mathbb{N}} F^n_\omega(S^\omega)$$
Infinite traces semantics as a fixpoint

Let $F_\omega$ be the function defined by:

$$F_\omega : \mathcal{P}(\mathcal{S}_\omega) \rightarrow \mathcal{P}(\mathcal{S}_\omega)$$

$$X \mapsto \{\langle s_0, s_1, \ldots, s_n, \ldots \rangle | \langle s_1, \ldots, s_n, \ldots \rangle \in X \land s_0 \rightarrow s_1\}$$

Then, $F_\omega$ is $\cap$-continuous and thus has a greatest-fixpoint; moreover:

$$\text{gfp}_{\mathcal{S}_\omega} F_\omega = [S]_\omega = \bigcap_{n \in \mathbb{N}} F_\omega^n(\mathcal{S}_\omega)$$

Proof sketch:

- the $\cap$-continuity proof is similar as for the $\cup$-continuity of $F_\star$
- by the dual version of Kleene’s theorem, $\text{gfp}_{\mathcal{S}_\omega} F_\omega$ exists and is equal to $\bigcap_{n \in \mathbb{N}} F_\omega^n(\mathcal{S}_\omega)$, i.e. to $[S]_\omega$ (similar induction proof)
Fixpoint form of the infinite traces semantics: iterates

**Example**, with the same simple transition system:
- $S = \{a, b, c, d\}$
- $\rightarrow$ is defined by $a \rightarrow b$, $b \rightarrow a$ and $b \rightarrow c$

Then, the first iterates are:

\[
\begin{align*}
F_0^\omega(S^\omega) &= S^\omega \\
F_1^\omega(S^\omega) &= \langle a, b \rangle \cdot S^\omega \cup \langle b, a \rangle \cdot S^\omega \cup \langle b, c \rangle \cdot S^\omega \\
F_2^\omega(S^\omega) &= \langle b, a, b \rangle \cdot S^\omega \cup \langle a, b, a \rangle \cdot S^\omega \cup \langle a, b, c \rangle \cdot S^\omega \\
F_3^\omega(S^\omega) &= \langle a, b, a, b \rangle \cdot S^\omega \cup \langle b, a, b, a \rangle \cdot S^\omega \cup \langle b, a, b, c \rangle \cdot S^\omega \\
F_4^\omega(S^\omega) &= \ldots
\end{align*}
\]

**Intuition**

- at iterate $n$, prefixes of length $n + 1$ match the traces in the infinite semantics
- only $\langle a, b, \ldots, a, b, a, b, \ldots \rangle$ and $\langle b, a, \ldots, b, a, b, a, \ldots \rangle$ belong to all iterates
Outline

1. Transition systems and small step semantics
2. Traces semantics
3. Summary
We have discussed:

- **small-step / structural operational semantics**: individual program steps
- **big-step / natural semantics**: program executions as sequences of transitions
- their **fixpoint definitions** and properties

**Next lectures:**

- another family of semantics, **more compact and compositional**
- **semantic program** and **proof methods**